AFRL-MN-EG-TP-2007-7415

A C++ ARCHITECTURE FOR UNMANNED AERIAL VEHICLE SIMULATIONS

Peter H. Zipfel
Air Force Research Laboratory
Munitions Directorate
AFRL/MNAL
Eglin AFB, FL 32542-6810

SEPTEMBER 2007

CONFERENCE PAPER AND BRIEFING CHARTS

AIR FORCE RESEARCH LABORATORY, MUNITIONS DIRECTORATE

■ **Air Force Material Command** ■ **United Air States Force** ■ **Eglin Air Force Base**

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* XX-09-2007	**2. REPORT TYPE** CONFERENCE PAPER AND BRIEFING CHARTS	**3. DATES COVERED** *(From - To)*

4. TITLE AND SUBTITLE A C++ ARCHITECTURE FOR UNMANNED AERIAL VEHICLE SIMULATIONS	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER 62602F
6. AUTHOR(S) Peter H. Zipfel	**5d. PROJECT NUMBER** 2068
	5e. TASK NUMBER 50
	5f. WORK UNIT NUMBER 06
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Research Laboratory, Munitions Directorate, AFRL/MNAL, Eglin AFB, FL 32542-6810	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFRL-MN-EG-TP-2007-7415
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory, Munitions Directorate, AFRL/MNAL, Eglin AFB, FL 32542-6810	**10. SPONSOR/MONITOR'S ACRONYM(S)** AFRL-MN-EG
	11. SPONSOR/MONITOR'S REPORT NUMBER(S) SAME AS BLOCK 8

12. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRIBUTION A: Approved for public release; distribution unlimited.
Approval Confirmation #AAC/PA 04-04-07-236; dated 4 April 2007.

13. SUPPLEMENTARY NOTES

See 'cover page' for pertinent metadata information.

14. ABSTRACT

The C++ computer language is well suited to model multi-vehicle engagements. Its prowess is exemplified by the conversion of a unmanned aerial vehicle simulation from FORTRAN to C++. The new architecture accommodates besides UAVs and moving targets also targeting satellites. Its class structure is outlined, and the communication bus between the encapsulated vehicle-objects is discussed. A generic UAV model with five degrees-of-freedom fidelity is used to demonstrate the interactive features of the simulation. Our experience has shown that C++ is the programming environment of choice for networked simulations.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			PETER H. ZIPFEL
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	SAR	32	**19b. TELEPHONE NUMBER** *(Include area code)*

A C++ Architecture for Unmanned Aerial Vehicle Simulations

AIAA Infotech @ Aerospace 2007, 7-10 May 2007, Rohnert Park, CA

Peter H. Zipfel, Ph.D.
U.S. Air Force Research Laboratory
Eglin AFB, FL 32542

Abstract

The C++ computer language is well suited to model multi-vehicle engagements. Its prowess is exemplified by the conversion of a unmanned aerial vehicle simulation from FORTRAN to C++. The new architecture accommodates besides UAVs and moving targets also targeting satellites. Its class structure is outlined, and the communication bus between the encapsulated vehicle-objects is discussed. A generic UAV model with five degrees-of-freedom fidelity is used to demonstrate the interactive features of the simulation. Our experience has shown that C++ is the programming environment of choice for networked simulations.

Introduction

In today's network-centric world, aerospace vehicles interact with many objects. They navigate by overhead satellites, synchronize their flight paths with other vehicles, swarm over hostile territory and attack multiple targets. Studying these engagements has become an important task of M&S. Engineers and analysts are using many environments, from the venerable FORTRAN language, the symbolic translators like MATRIX$_X$™, MATLAB™, and VisSim™, to the newer languages of C and C++. The multi-object environment of network-centric engagements is particularly well supported by the object oriented computer language C++. Hence, we see with increasing frequency conversions of legacy code and new simulations coded in C++.

The Unmanned Aerial Vehicle (UAV) simulation at the Air Force Research Lab was converted from FORTRAN to a new C++ architecture, called CADAC++, which makes possible multiple instantiation of UAVs, targets, and satellites. This new capability enables the study of fly-out trajectories, third party targeting, and distributed information sharing. The **objective** of this paper is to highlight the simulation architecture and to present some typical engagement scenarios.

The architecture is based on the hierarchical structure of inherited classes. The UAV, target, and satellite classes, inherit the three degrees of freedom equations of motion from the classes `Round3`, conveying the spherical rotating Earth model. In turn, these classes inherit the communication structure from the base class `Cadac`. The components of the vehicle, e.g., aerodynamics, propulsion, and autopilot, are represented by modules, which are member functions of the vehicle classes. Communication among the modules occurs by protected module-variable arrays. Every instantiated vehicle object is encapsulated with its methods and data. To communicate between vehicles, data packets are loaded onto a global data bus for recall by other vehicles. Input occurs by ASCII file and output is compatible with CADAC Studio, a plotting and data processing package.

The UAV simulation is chiefly a synthesis tool for refining the components of the primary vehicle and exploring its performance as it interacts with its environment. Its three translational degrees of freedom are augmented by pitch and bank angle dynamics (a so-called pseudo 5 DoF simulation). Autopilot functions are modeled by transfer functions, which generate the inner-loop dynamics, while the outer loop contains the navigation and guidance functions. A terminal seeker guides the vehicles into the target with proportional navigation.

The **paper** outlines the class structure of the simulation, address the polymorphism that creates the vehicle objects during run-time, and explain the communication bus amongst the objects. Key components of a generic UAV – aerodynamics, propulsion, guidance and control – are summarized, followed by trajectories plots of UAVs, satellites, and targets.

Requirements

CADAC++ is chiefly an engineering tool aiding in the development of aerospace vehicles. Though it focuses on the main vehicle – UAV, aircraft, or missile – it also portrays the interactions with outside elements, like satellites, targets, and sister vehicles. The main vehicle is modeled with greatest fidelity, while the secondary objects have simpler representations.

The synthesis and conceptualization process places distinct requirements on the simulation architecture . To support the design engineer in evaluating the aerodynamics, propulsion, guidance and control components, CADAC++ should mirror the same modular structure and closely control the interfaces between them. It should encapsulate each vehicle object for multiple instantiation and provide global communication between them. Input and output must be flexible and compatible with CADAC Studio, a post processing and analysis tool. More specific requirements follow.

Face to the User

The user likes to focus on the evaluation of the main vehicle without being burdened by the details of the simulation's execution. He wants control over the input/output and the vehicle modules that code the subsystems.

There should be only one input file that controls the simulation. It displays the run title, an option line for directing the output, the calling sequence of the modules, the sizing of the integration step and output intervals, and the initializing of the vehicle parameters. The aerodynamic and propulsion tables should be kept separate for safekeeping. Therefore, in the input file, there would be only provided the files name of the tables. Multiple instantiation of the vehicle objects should be accomplished by simply duplicating the vehicle data and possible variations to the input parameters.

The output control should be simple yes/no choices. An option line would provide output to the screen of the primary and secondary vehicles together with the event messages that indicate their changing flight status. There should also be an option to archive the screen output to a file. Plot files would be written for individual vehicles and merged together for multi-vehicle displays. These output files should be compatible with CADAC Studio for two and three dimensional plotting.

The components of the vehicles should be mirrored by modules containing code that models their features. Strict control of the interfaces will make the modules interchangeable amongst simulations. The modules should define these interface variables, execute integration of state variables and enable table look-up. Any vehicle changes that the user has to make should be confined to these modules.

Multiple Encapsulated Vehicle Object

Each aerospace vehicle, be it UAV, aircraft or missile, should be built up from a hierarchy of classes, starting with the base class `Cadac`, followed by the equations of motion, and completed by the vehicle itself. Each vehicle is a C++ object with its data (aerodynamics and propulsion) and methods (modules) encapsulated. Run-time *polymorphism* should be used to sequence through the vehicles objects during execution.

Modularity of Vehicle Components

The modules, representing the vehicle components, should be *public* member functions of the vehicle classes. Their interfaces – the module-variables – would be stored in *protected* data arrays that are available to all modules of the vehicle object. During execution, the modules should define all module variables, make initializations, integrate state variables, and conduct post run calculations.

Event Scheduling

Just as aerospace vehicles transition though flight phases, the simulation should be able to sequence through such events. These events should be controlled by the input file without any code changes in the modules. Relational operators such as <, =, > would be applied to the module-variables and trigger the events.

Global Communication Bus

Because vehicle objects are encapsulated into classes, a global communication bus should enable the transfer of data. Each vehicle should be able to *publish* and *subscribe* to any of the module-variables.

Table Look-up

Table utilities should provide for one, two, and three independent variable look-up. Tables must be stored in separate files and modifications easily accomplished. Simple syntax should make the table look-up easy to program in the modules.

Matrix Utility Operations

The full power of C++ should be applied to matrix operations. Matrix utilities should be tailored to the specific needs of flight simulations and not burdened by C++ *container* classes. Efficient pointer arithmetic will speed up the execution and will allow unlimited sequencing of matrix operations.

Documentation and Error Checking

The module-variables, being the key interfaces between the modules, should be completely documented. Their definitions, provided in the modules, should be collected in a single output file. The module-variables in the input file should also be documented with the same definitions.

Error checking should identify module-variables that have not been assigned the correct names or locations in the input file or the modules. Incompatible matrix operations should be flagged, as well as problems with opening of file streams.

Architecture

All these requirements can be satisfied with object oriented programming in C++. Hierarchical class structures, encapsulation of data and methods, run-time polymorphism, overloading of functions and operators, are all features used in CADAC++ to build a simulation environment suitable for flight vehicle synthesis.

CADAC++ programming follows strictly the International Standard for C++, defined by the ANSI/ISO Committee in 1998 and implemented by most compilers like Microsoft Visual C++. Thus portability is assured and low cost operation is made possible.

Each requirement is now addressed separately with particular focus on the classes that structure the features of CADAC++, see Fig. 1

CLASS	DESCRIPTION
Cadac, ...	Hierarchical class structure of vehicles
Vehicle	Hosting a pointer array of type **Cadac**
Module	Storing module information
Variable	Declaring module-variables
Event	Storing event information
Packet	Declaring data packets for global communication bus
Datadeck	Hosting a pointer array of type **Table**
Table	Storing tabular data
Matrix	Storing matrix operations
Document	Storing module-variable definitions

Fig. 1 CADAC++ Classes

Multiple Encapsulated Vehicle Object

The rewriting of CADAC was motivated by the unique feature of C++ allowing encapsulation of vehicle objects. Encapsulation means binding together data and functions and restricting their access. The aerodynamic and propulsion data are bound together with the table look-up functions and many other functions that support the missile and aircraft objects. In turn, these objects are created from a hierarchical class structure derived from the common base class **`Cadac`**.

This hierarchical class structure in CADAC depends on the particular simulation. For instance, the UAV simulation consists of three branches `Cadac` ← `Round3` ← `Cruise`, `Cadac` ← `Round3` ← `Satellite`, and `Cadac` ← `Round3` ← `Satellite`, , where `Round3` models the equations of motions over the spherical Earth, and `Cruise`, `Target`, and `Satellite` modes the components of the vehicles.

The vehicle objects, declared by their respective classes, are created during run-time by the polymorphism capability of C++. Polymorphism – many forms, one interface – uses inheritance and virtual functions to build one vehicle-list of all vehicle objects, be they UAVs, targets, or satellites. At execution, this vehicle-list is cycled through each integration step in order to compute the respective vehicle parameters.

Through run-time polymorphism any number of different vehicles can be called using the common pointer array of type `Cadac`. These calls are executed during initialization and at every integration step. A limitation of this architecture is that all vehicle objects have to be instantiated at the beginning of the run.

Modularity of Vehicle Components

A key feature of CADAC is its modularity, reflecting the component structure of an aerospace vehicle. Just as the hardware is divided into subsystems like propulsion, autopilot, guidance, and control, so is the CADAC simulation broken into propulsion module, autopilot module, etc., and the more esoteric modules like aerodynamics, Newton's equations of motions, and environment. This one-for-one correspondence ensures clean interfaces between the modules.

Each module is a pure virtual member function of the abstract base class `Cadac` and is overridden in the derived class, be it `Round3`, `Cruise`, `Target`, or `Satellite`. If the derived class does not use a module, the module will return without code.

The calling sequence of the modules is controlled by their sequential listing in the input file `input.asc`. Each module may consist of four parts: the definition part identified by `def`, the initialization part, `init`, the execution part, `exec`, and the last call, `term`. All but the `exec` part are called only once, `exec` is called every integration step.

Module-variables provide the sole data transfer between the modules of a vehicle object. For documentation they are recorded in sequential order in `doc.asc` with their definitions and other relevant information. Between their label and array location, there is a unique one-on-one relationship. Any deviation from that rule is flagged in `doc.asc`.

Event Scheduling

As aerospace vehicles fly their trajectories, they may sequence through several events towards their destinations. Just think of rockets staging; airplanes taking off, cruising and landing; and missiles passing through midcourse and terminal phases towards the intercept. Events in CADAC++ are interruptions of the trajectory for the purpose of reading new values of module-variables. They can only be scheduled for the main vehicle object. The maximum number of events is determined by the global integer `NEVENT`, while the number of new module-variables in each event is limited by the global integer `NVAR`.

An event is defined in the input file `input.asc` by the event block starting and ending with the key words `IF` … `ENDIF`. Appended to `IF` is the event criterion. It consists of the watch variable – any module-variable, except of type Matrix – and a relational operator followed by a numerical value. For instance,

```
IF dbt < 8000
   mseek 2
ENDIF
```

meaning, if the range to the target is less than 8000 m, the seeker is enabled (mseek=2). The supported relational operators are `<, =, >.`

The **`Event`** class supports the creation of `Event` type objects. The private members of the `Event` class store information about the event, such as watch variable, relational operator, threshold value, and new module-variables. The public methods are `set` and `get` functions for the data. To expedite execution, the new module-variables are not stored by their name, but by their offset index in the module-variable array. Therefore, rather than cycling through all the module-variables, the new module-variables are directly picked out by their offset indices. These index lists are also part of the private data members of `Event`.

Event scheduling gives great flexibility to shaping the trajectory of an aerospace vehicle. However, as a design matures and the switching logic becomes well defined, the events can be scheduled in the module itself, and any event scheduling in the `input.asc` file may be completely eliminated.

Global Communication Bus

Encapsulation by classes isolates vehicles objects from each other. This great feature of C++, however, prevents direct communication between the vehicles. For instance, the missile object needs to know the coordinates of the target object in order for its seeker to track it. How can the missile get access to the protected target data?

In CADAC++ the global communication bus, called **`combus`**, provides this interface. Selected module-variables are stored in `combus` so that other vehicles can download them. To identify this process we borrow the terms '**publish**' and '**subscribe**' from HLA (High Level Architecture).

The enabling global class is **`Packet`**, which, as a private data member, stores the vehicle ID, the status of the vehicle (alive, hit, dead), the number of module-variables in the data set, and a pointer to the array of module-variables of type **Variable**. Each vehicle object contributes one packet to the communication array combus of type Packet. The slot # is the same as that of the vehicle in the vehicle_list.

Each packet has a data set of module variables stored in the array, pointed to by Variable *data. The storage sequence in the data set is in the order the module-variables are read, which is given by the module sequence in the input file input.asc. This sequence is important for the subscription process.

The subscription of module-variables occurs in the modules. For instance, the seeker in order to track the target has to subscribe to the target coordinates. First, the target ID is built from the string "t" and the tail number of the target. Then combus is searched for this packet and the data set is downloaded

The number of module-variables in the data set is unrestricted. If you are unsure of the storage sequence, you can find it by selecting y_comscrn and counting the labels, just make sure that you count the three components of vectors as one label only.

Table Look-up

Interpolating aerodynamic and propulsion tables is an important task in any aerospace simulation. Aerodynamic coefficients are usually given as functions of incidence angles and Mach number; sometimes also as a function of altitude and control surface deflections. Propulsion data are tailored to the type of propulsion system. For rocket motors, simple thrust tables may suffice, while turbojet and ramjet engines depend on throttle, Mach number, and, for more accurate models, even on incidence angles.

The more independent variables are included, the higher the complexity of the table. Seldom, however, is the dimension higher than three – dictated by runtime considerations. CADAC++ supports table look-up schemes up to third dimension and interpolates linearly between the discrete table entries. It keeps the so-called 'data decks' as separate files, so they can be properly protected, as the need may arise. If any changes have to be made – adding or deleting tables – they are absorbed automatically.

The handling of the tables is accomplished by the two classes: Datadeck and Table. The class **Datadeck** has a private member **table_ptr, which is a pointer to an array of pointers of the class **Table** that contains the pointers to all the tables of a data deck, be it the aerodynamics or propulsion deck. Under the 'main vehicle' scope, inside the 'protected' access specifier, the objects Datadeck aerotable and Datadeck proptable are declared, and also the table pointer Table *table. At execution, two distinct phases take place: loading the tables and extracting the interpolated value.

Additions and deletions of tables in the AERO_DECK or PROP_DECK are automatically adjusted during the loading of the tables. If a simulation requires data tables of a different type – e.g., antenna pattern – , one has to do four things: (1) create an ASCII file with the data tables, (2) identify the file name by a key word – ANT_DECK antenna_data.asc – in the input.asc file, (3) declare an additional Datadeck object in the 'main vehicle' class – antennatable –, and (4) replicate in the function input_data(…) a third block for the new key word.

Matrix Utility Operations

Modern programming makes use of matrix operations as much as possible. It condenses code and avoids errors caused by coordinating equations. CADAC++ has a rich set of matrix operations, which are public members of the class **Matrix**. This class is tailored to the special needs of flight dynamics. Generality has been sacrificed for efficiency. Rather than using template classes and particularly the vector container class of the STL, the CADAC++ matrix operations are restricted to variables of type double.

The **Matrix** class declares a private pointer to the matrix array double ***pbody;** together with the array dimensions. There are 48 matrix operations declared in the public access area. They can be divided into 30 functions and 18 overloaded operators.

The matrix utilities have a full suit of overloaded operators. The assignment operator requires a **copy constructor** to provide for a deep copy of the object to assure that the new object has its own memory allocated, and that it is recoverable when the object is destroyed.

The offset operator `[]` is also overloaded to access the elements of a `Matrix` array. However, this works only for one-dimensional arrays, because two-dimensional arrays require more than one offset operator. For those, the Matrix functions `assign_loc(..)` and `get_loc(…)` must be used.

Documentation and Error Checking

Self-documentation is an essential part of any simulation. Of primary interest are the variables that are used for input/output, as interfaces between modules, and those of particular interest for diagnostics. All are referred to as **module-variables**. The description of a module-variable occurs only once, in the 'def_module' function. This description is used to document the input file `input.asc` and to create a list of all module-variables in the output file `doc.asc`. The documentation of `input.asc` is automatic, while the `doc.asc` file is only created if the `OPTION y_doc` is selected.

Error checking focuses in particular on the correct formatting of the `input.asc` file and the enforcement of the one-on-one correspondence rule: "One module-variable name for one array location". Other checks assure that matrix operations are performed on compatible matrices and that file streams open correctly.

The class **`Document`** is used to make the module-variable descriptions available. Its private data are essentially a subset of the class `Variable`. They store name, type, definition and module of each module-variable.

During initialization, a check is made whether that slot is empty and can receive a new variable. If not, the error code '*' is set. As the function `document()` writes the output file `doc.asc`, the module-variable array is checked for duplicate names. The error code 'A' is set if this occurs. Both codes are inserted in the first column of the `doc.asc` file and a warning message is sent to the console.

A good description of a particular simulation is produced if the modules, the `input.asc`, and the `doc.asc` files are collected in a document. It should enable someone else, who is familiar with the CADAC++ framework, to pick up, run, and understand the simulation.

Aerial Vehicle Model

The generic UAV that is used in the simulation is shown in Fig. 2

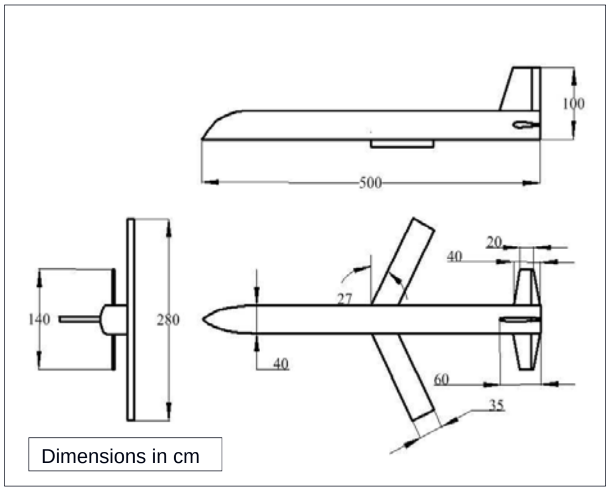


Fig. 2 Layout of UAV

It's aerodynamics is obtained from DATCOM and modeled as drag polars, see Fig. 3

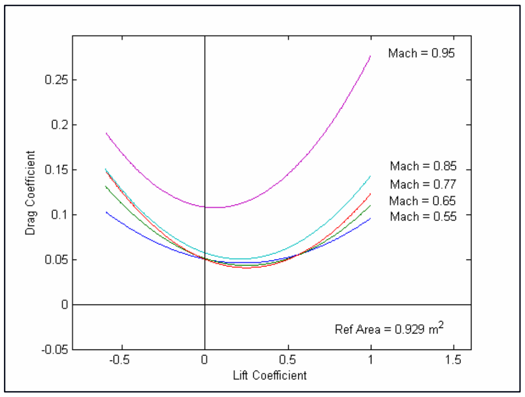


Fig. 3 Drag polar

The propulsion data reflect a typical turbojet with data tables as shown in Fig. 4

- Thrust available = fct (Mach, alt)
- Fuel flow = fct (Mach, Alt)
- Idle thrust = fct (Mach, alt)
- Idle fuel flow = fct (Mach, alt)

Fig. 4 Propulsion data

A typical thrust available diagram is shown in Fig. 5

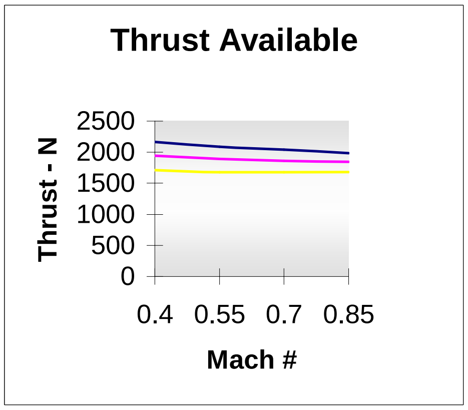


Fig. 5 Thrust available for altitudes 0, 1524, and 3048 m

During cruise the UAV must be able to maintain constant speed, so a Mach hold controller is implemented as shown in Fig. 6

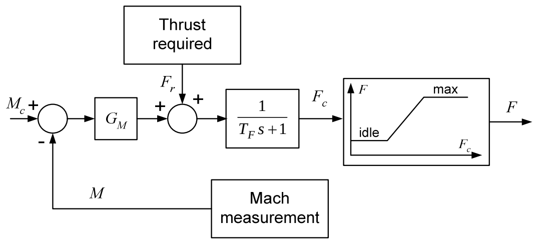


Fig. 6 Mach hold controller

The autopilot consists of multiple controllers. The autopilot location is shown in the block diagram of Fig. 7.

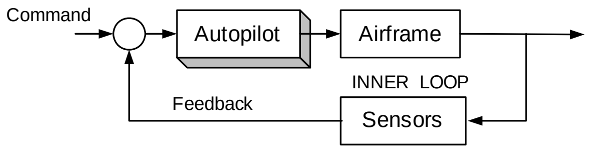


Fig. 7 Autopilot

To serve all the flight phases it must have several modes as summarized in Fig. 8

- Bank angle control
- Flight path angle control
- Heading control
- Normal acceleration control
- Lateral acceleration control
- Altitude control

Fig. 8 Autopilot modes

The UAV steers from waypoint to waypoint using onboard guidance, see Fig. 9

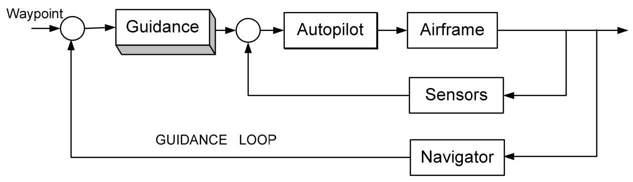


Fig. 9 Guidance

Besides waypoint guidance, the UAV also has the capability to attack a target either using its on-board seeker, or obtaining the target coordinates from satellites.

Engagement Scenarios

Three scenarios demonstrate the capability of the UAV netcentric simulation. The first scenario, Fig. 10, depicts a UAV flying through three waypoints approaching the target area and homing into the target autonomously with its on-board seeker.

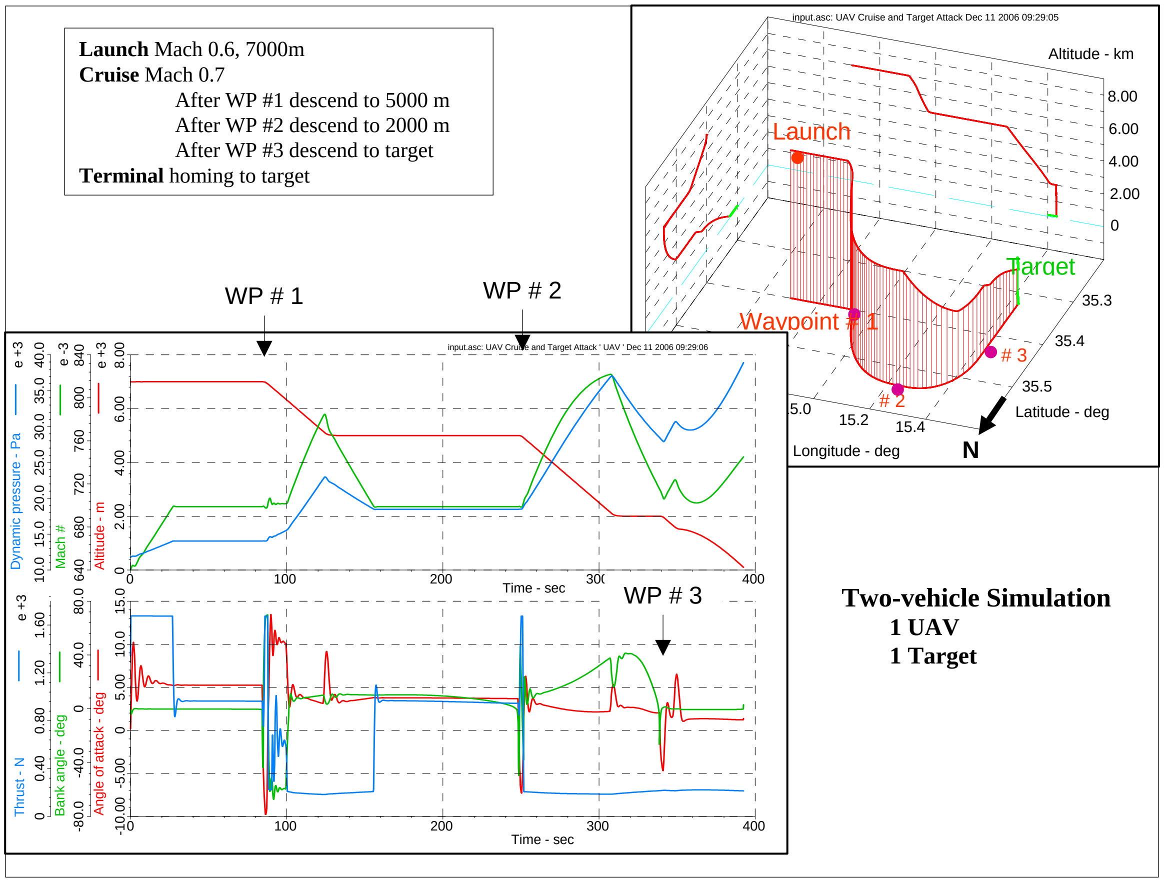


Fig. 10 One-on-one engagement with terminal seeker guidance

In the simulation, the coordinates of the target are published to the `combus` communication bus and subscribed by the UAV. Thus the seeker is pointed at the target and can provide the line-of-sight rates to the guidance computer for an intercept.

In the second scenario, Fig. 11, instead of the seeker guiding the vehicle, the target coordinates are sent by an overhead satellite to the UAV for tracking.

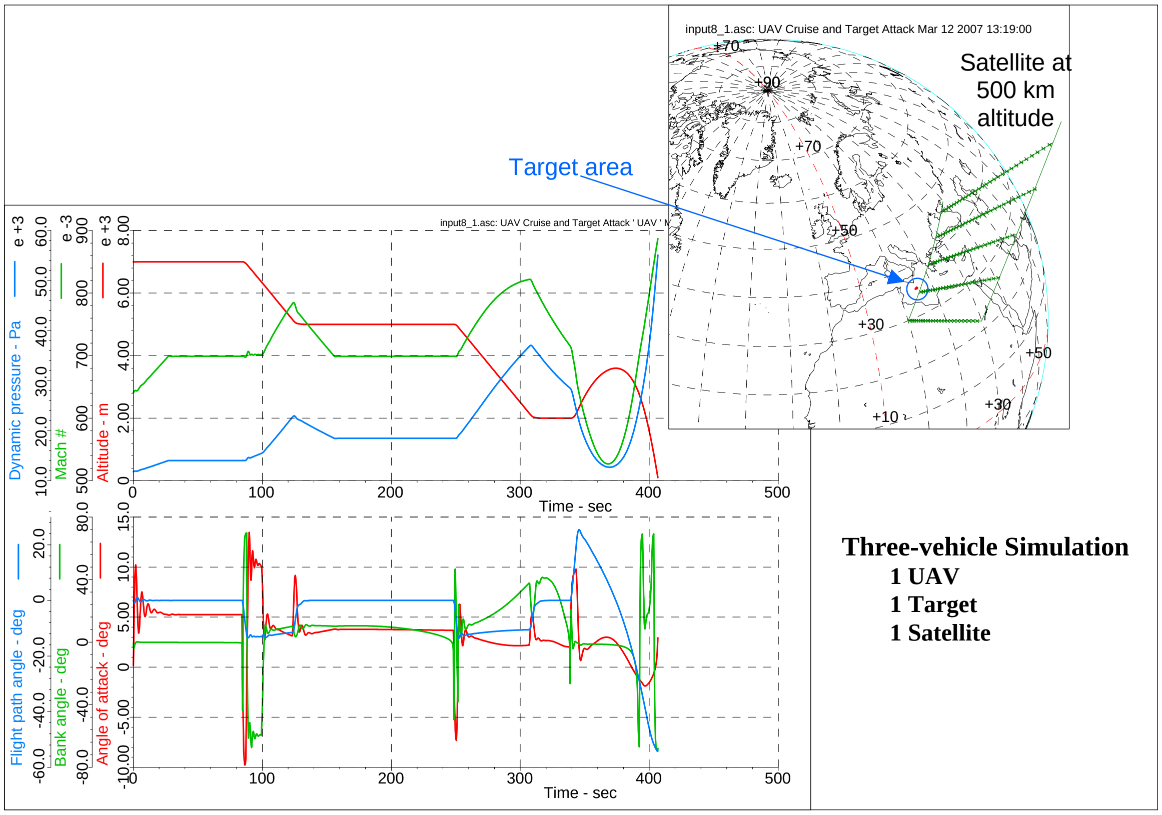


Fig. 11 One-on-one engagement with satellite targeting

In this engagement there are three vehicle objects active. The UAV approaches the target area via waypoints, while the overhead satellite tracks the moving target on the ground and relays the target coordinates to the UAV guidance processor.

In the simulation, all three vehicle objects publish their coordinates to `combus`. They are used to make visibility calculations, i.e., can the satellite see the target and is there a clear line-of-sight to the UAV for broadcasting the target coordinates. If affirmative, the UAV subscribes to the target coordinates from `combus` and makes the intercept.

In the third scenario, Fig. 12, nine vehicle-objects are simulated. The three UAVs attack three targets, while three satellites orbit the Earth. Two of the UAVs are guided by their on-board seeker, while the third one receives targets coordinates from Satellite #1 that is closest to the target area.

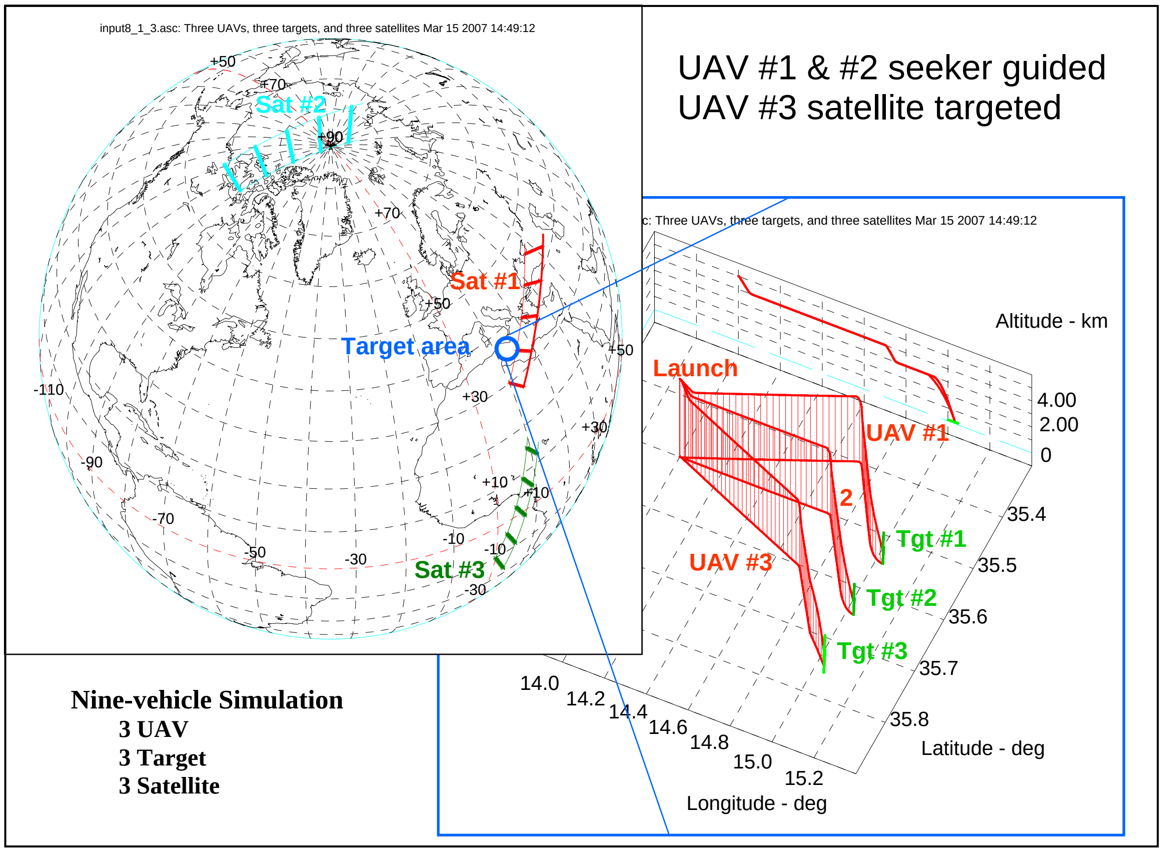


Fig. 12 Three-on-three engagement with seeker and satellite targeting

In the simulation, `combus` keeps track of nine vehicle-objects. Line-of-sight calculations are carried out and the three UAVs subscribe to three target coordinates for intercept either with their onboard seeker or by using the satellite supplied target coordinates.

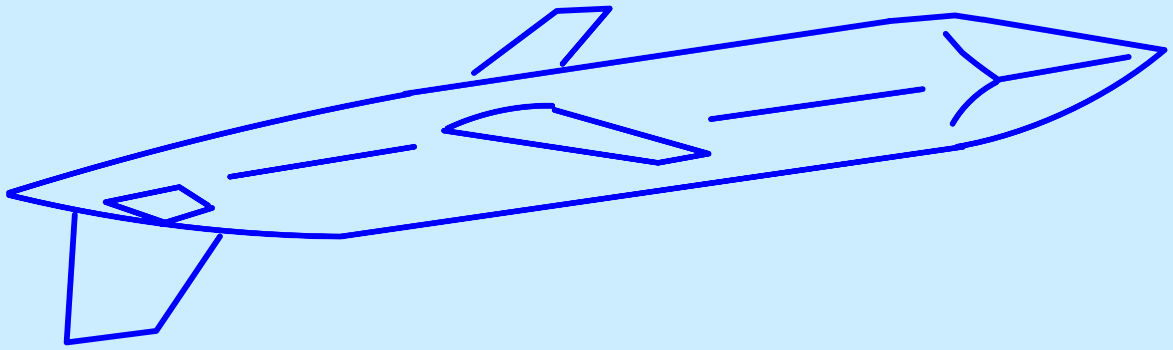
Conclusions

The conversion of the CADAC unmanned aerial vehicle simulation from FORTRAN to C++ has been completed successfully. What used to be a one-on-one simulation has now become a multiple engagement simulation thanks to object oriented paradigm of C++. Our experience has shown that C++ is the programming environment of choice for networked simulations, outperforming Matlab/Simulink-based simulations in programming, execution speed, and cost.

A C++ Architecture for Unmanned Aerial Vehicle Simulations

Peter H Zipfel, Ph.D.

zipfel@eglin.af.mil



AIAA Infotech @ Aerospace 2007, 7-10 May 2007, Rohnert Park, CA

Overview

- Requirements for a Netcentric Simulation
- UAV Netcentric Simulation
- CADAC++ Class Structure
- Class Hierarchy of UAV Simulation
  - Implementation of Run-Time Polymorphism
  - CADAC Modularity
  - Event Scheduling
  - Global Communication Bus
  - Matrix Utility Operations
  - Documentation and Error Checking
- UAV with Waypoint Navigation and Seeker Homing
- UAV Receiving Target Coordinates from Satellite
- Netcentric Engagement
- Summary

Requirements for a Netcentric Simulation

- Synthesis capability for multi-vehicle environments
  - Higher fidelity simulation of main vehicle
  - Lower fidelity for supporting vehicles
- Encapsulation of vehicles for multiple instantiation
  - Binding data and functions and restricting their access
- Modular structure to mirror the vehicle's components
  - Strict interface control
  - Re-use of code
- Event scheduling
  - Simulating the phases of flight
- Global communication bus
  - Data flow between encapsulated objects
- Table look-up
  - 1, 2, 3 – dimensional
  - Data decks separated for safekeeping
- Matrix utility operations
  - Combining matrix operations like scalars
- Documentation and error checking
  - Documenting all interface variables
  - Checking interface variables, matrix compatibility, and file streams
  - Output compatible with CADAC-Studio ♣

CADAC++ UAV Simulation

Compatibility

Microsoft Visual C++ 8

CADAC Studio plotting (IBM PC, Windows)

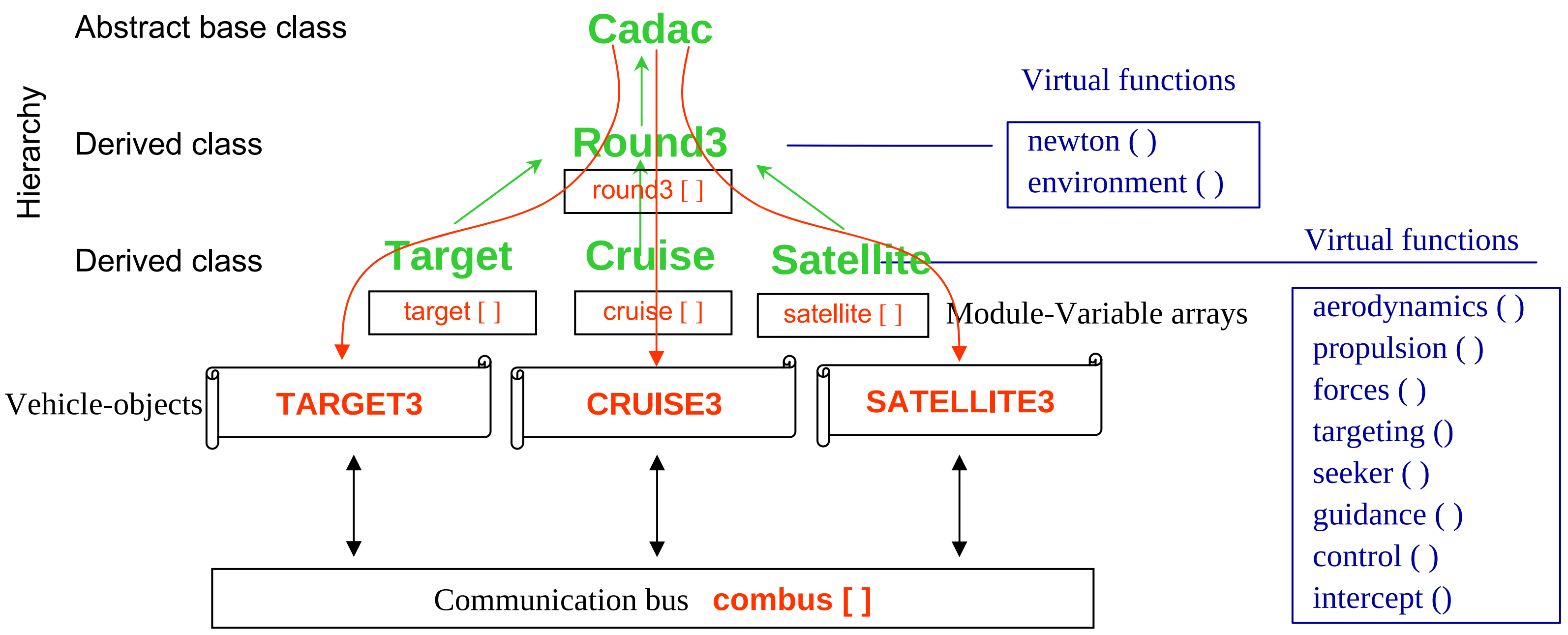
- **5 DoF spherical rotating Earth**
  - **UAV**
    - **3 translational DoF**
    - **2 rotational DoF: pitch and bank**
  - **Target**
    - **Moving on ground**
  - **Satellite**
- **UAV**
  - **Aerodynamics**
    - **Trimmed**
    - **Drag polar function of Mach**
  - **Propulsion**
    - **Turbojet**
    - **Mach controller**
  - **Flight controllers**
    - **Bank angle**
    - **Flight path angle**
    - **Heading**
    - **Altitude**
    - **Acceleration**
  - **Guidance**
    - **Waypoint**
      - **Point**
      - **Line**
      - **Arc**
    - **Terminal**
      - **Pro-nav**
      - **Line guidnce**
  - **Seeker**
    - **Simple line-of sight**
  - **Satellite targeting** ♣

CADAC++ Class Structure

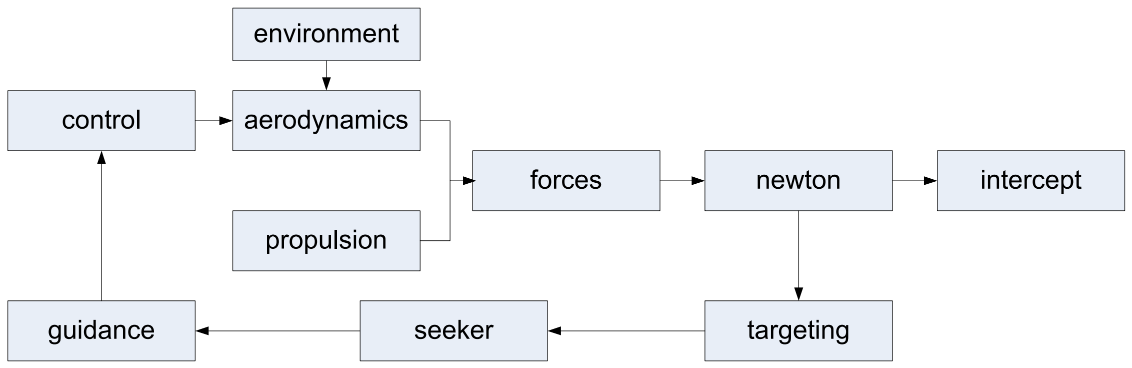
CLASS	DESCRIPTION
Cadac, ...	Abstract base class of hierarchical structure
Vehicle	Hosting a pointer array of type **Cadac**
Module	Storing module information
Variable	Declaring module-variables
Event	Storing event information
Packet	Declaring data packets for global communication bus
Datadeck	Hosting a pointer array of type **Table**
Table	Storing tabular data
Matrix	Storing matrix operations
Document	Storing module-variable definitions

♣

Class Hierarchy of UAV Simulation



Modular Structure



Implementation of Run-Time Polymorphism

- All vehicle objects are stored in a pointer array `vehicle_ptr` of type `Cadac`
  1. Create `Vehicle vehicle_list` which has as private member the pointer array `Cadac **vehicle_ptr`
  2. From 'input.asc' read the number and type of vehicle objects
  3. Add the vehicle pointers to `vehicle_ptr` array in the order read from 'input.asc'
- During run-time the vehicle objects are accessed by their pointers
  - The class 'Vehicle' declares an overloaded offset operator `Cadac *operator[]` that returns the vehicle pointer
  - The vehicle pointer is of the correct vehicle type (e.g., `Cruise`, `Target`, `Satellite`) although it is stored in the pointer array of type `Cadac` (polymorphism)
  4. With this vehicle-pointer access the member functions of the respective vehicle
  - Example: At every integration step the 'newton' module of the i-th vehicle is called

```
vehicle_list[i]->newton(int_step);
```

♣

CADAC Modularity

- CADAC's modular structure mirrors the hardware components of an aerospace vehicle
  - A module is a model of a vehicle component
    - Examples: aerodynamics, propulsion, actuator, guidance, control,…
  - Each module consists of at least two functions and not more than four
    - `def_module(), init_module(), module(), term_module()`
- The calling order of the module is controlled by the input file
- Data between modules is transferred by module-variables
  - Module-variables, being the only allowed interface, are strictly controlled
  - Each vehicle object reserves protected arrays for its module-variables
  - There is a one-to-one relationship between the module-variable name and the array location
  - The file doc.asc documents all module-variables
- Module-variables can be of type int, double, 3x1 vector, and 3x3 matrix
- Inside a module
  - Module-variables are localized for input
  - Computations create other module-variables
  - These are loaded onto the object's array for output ♣

Event Scheduling

- **Vehicle trajectories are divided into phases initiated by events**
  - **Take-off, cruise, landing**
  - **Autopilot command changes**
  - **Guidance mode changes**
- **Events in CADAC++ are interruptions of the trajectory for the purpose of reading new module-variables**
  - **Global dimensioning of events**
    - **NEVENT = maximum number of events**
    - **NVAR = maximum number of module-variables in each event**
- **Events are introduced in the input file 'input.asc'**
  - **Event block**
    - **IF** ***watch_variable_name relational_operator value***
      - » new module-variables
    - **ENDIF**
  - **Supported relational operators < , = , >**
  - **Example: After 5 sec, altitude command is changed to 5000 m**

```
IF time > 5
     altcom  5000    //Altitude command  - m  module control
ENDIF
```

♣

Global Communication Bus

- **Encapsulation of vehicle-objects prevents direct data exchange**
- **The communication bus `combus` gives global access to the module-variables of all vehicle-objects**
- **Building the communication bus**
  - **Module-variables are flagged by the keyword "`com`"**

    ```
    flat6[56].init("vmach",0,"Mach number","environment","out","scrn,plot,com");
    ```
  - **Every vehicle-object publishes (loads) a packet of "`com`" –variables**
  - **The packets are stored in the array `combus` of type `Packet`**
- **Using the communication bus**
  - **The communication bus can be used by any vehicle-object**
  - **A vehicle-object subscribes (downloads) to the variables it needs from the other vehicle-objects**
  - **Example: UAV downloads the position of the target it attacks**
- **Characteristics of `combus`**
  - **It is an array of type `Packet` of size equal to the number of vehicle-objects**
  - **The slot # of a vehicle in `combus[]` is the same as in `vehicle_list[]`**

♣

Matrix Utility Operations

- Source code should be programmed in matrices as much as possible
  - Compact code
  - Avoids errors
- Requirements of flight simulations
  - Mostly 3x1 and 3x3 matrices, some of higher dimensions (Kalman filters)
  - Elements of matrices are of type `double`
- Class `Matrix` instantiate a pointer to the matrix `*pbody` and initializes the elements to zero
- Examples from module `Target::forces()`

```
//Coriolis acceleration in V-coordinates
WEIG=TGE*WEII*TEG;
CORIO_V=TVG*WEIG*VBEG*2;
//centrifugal acceleration in V-coordinates
CENTR_V=TVG*WEIG*WEIG*TGI*SBII;
```

- Special features
  - All matrix manipulations are carried out in pointer arithmetic
  - Creating a matrix (other than instantiation) returns `*this`, the re-created object
  - Copy constructor for the matrix assignment operator
  - Overloaded offset operator `[]` ♣

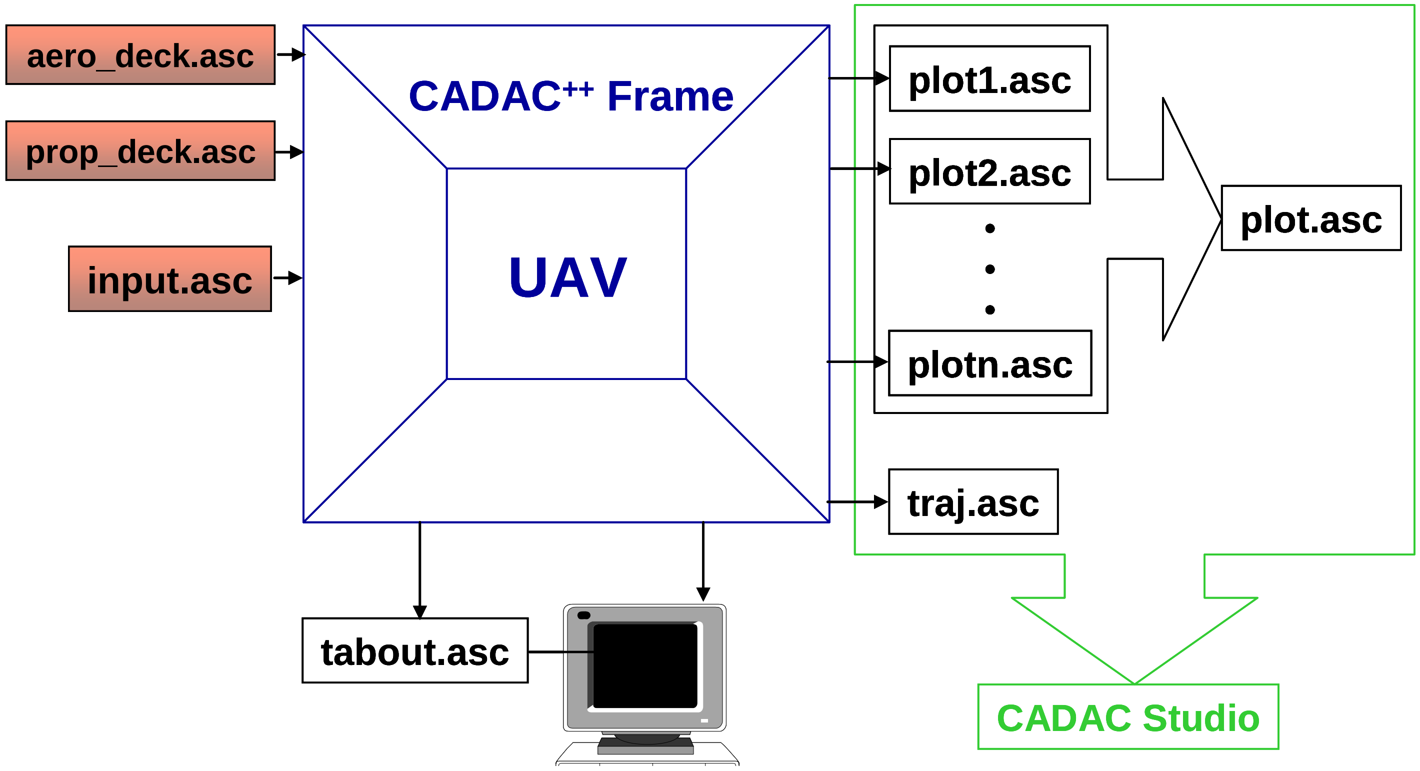
Documentation and Error Checking

CLASSES

- Cadac, ...
- Vehicle
- Module
- Variable
- Event
- Packet
- Datadeck
- Table
- Matrix
- Document

- **Emphasis is on documenting module-variables. They govern:**
  - **Input/output**
  - **Data transfer between modules**
  - **Special diagnostic needs**
- **'One definition – multiple use' principle**
  - **Module-variables are described in the modules**
  - **Their description is used in the `input.asc` file**
  - **All descriptions are collected in the `doc.asc` file**
- **Class `Document` enables the sharing of the descriptions**
- **Error checking**
  - **Matrix compatibility**
  - **File stream opening**
  - **Violations of the 'one-on-one correspondence' rule**
    - **One module-variable name for one array location**
- **Documentation package for a simulation**
  - **Modules**
  - **`input.asc`**
  - **`doc.asc`** ♣

Input/Output of CRUISE

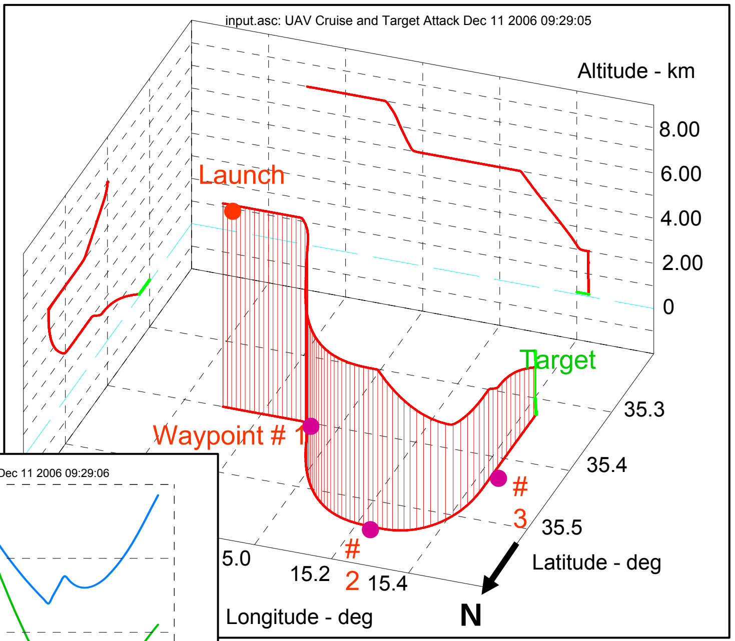


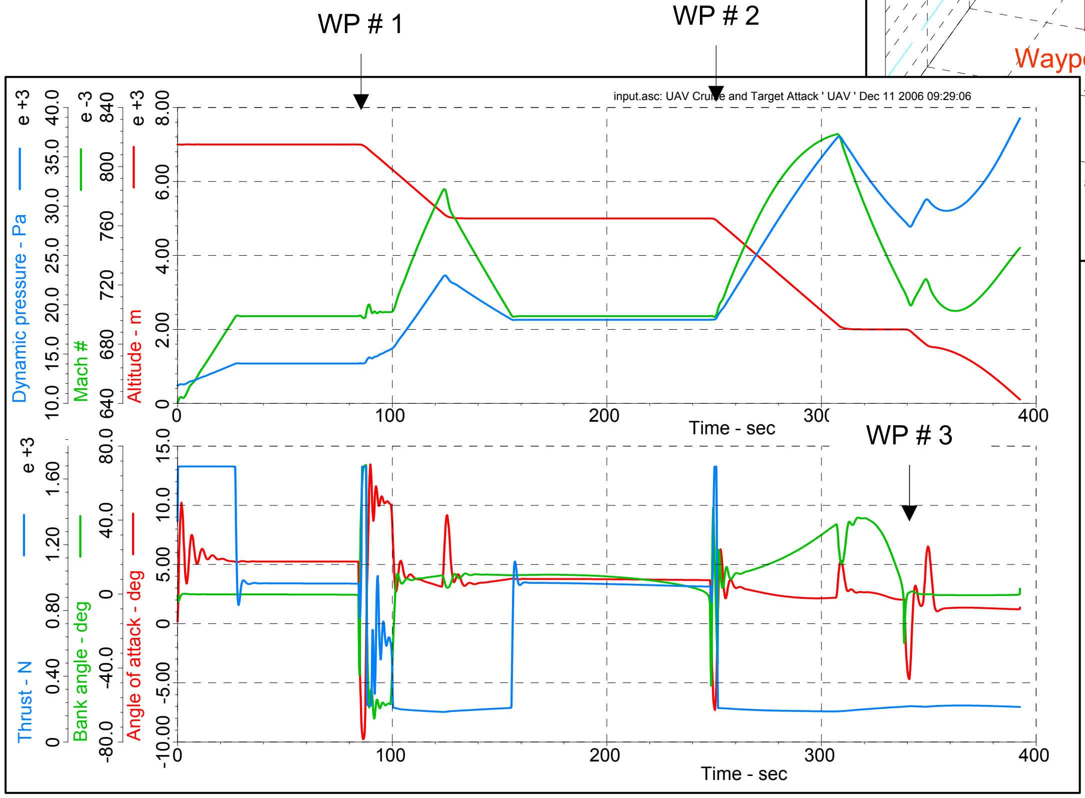
♣

UAV with Waypoint Navigation and Seeker Homing

Launch Mach 0.6, 7000m
Cruise Mach 0.7
- After WP #1 descend to 5000 m
- After WP #2 descend to 2000 m
- After WP #3 descend to target

Terminal homing to target

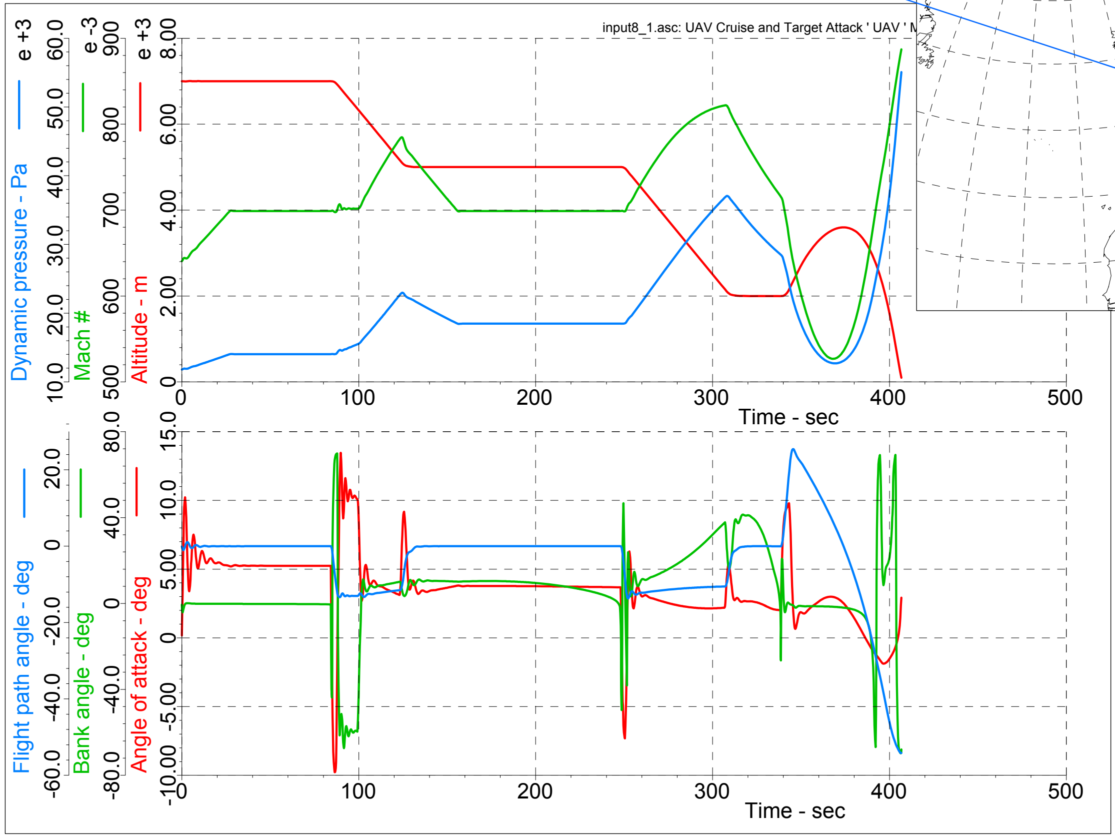


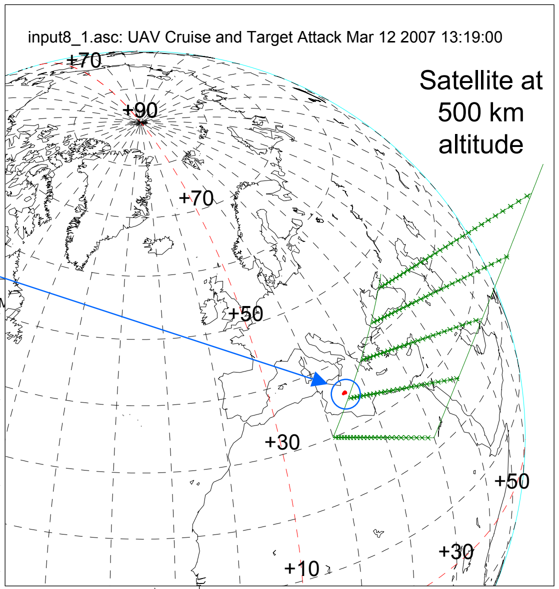


Two-vehicle Simulation
- **1 UAV**
- **1 Target**

UAV Receiving Target Coordinates from Satellite

- Seeker guidance replaced by satellite targeting




input8_1.asc: UAV Cruise and Target Attack Mar 12 2007 13:19:00
Satellite at 500 km altitude
Target area
+90
+70
+50
+30
+10

Three-vehicle Simulation

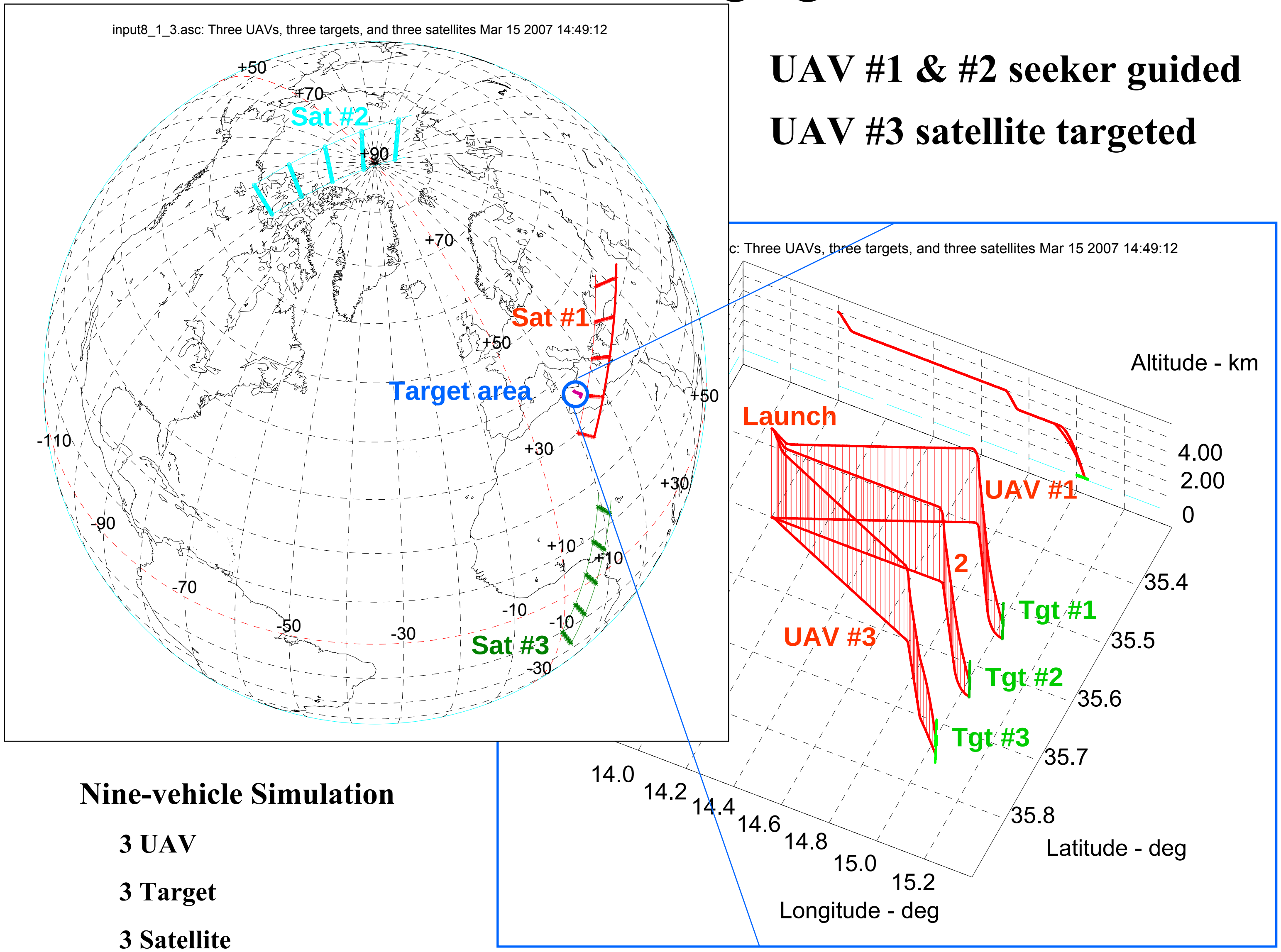
1 UAV

1 Target

1 Satellite

♣

Netcentric Engagement



UAV #1 & #2 seeker guided

UAV #3 satellite targeted

Nine-vehicle Simulation

- **3 UAV**
- **3 Target**
- **3 Satellite**

Other Architectures

- ENGAGE++ 3 DoF air-to-air engagement simulation
  - Air Force Research Lab
  - GUI directed
  - Partial hierarchical class structure, no abstract base class
- JSBSim FlightGear simulator
  - <u>J</u>on <u>S</u>. <u>B</u>erndt
  - Embedded in real time simulator
  - Partial hierarchical class structure, no abstract base class
- CMD <u>C</u>++ <u>M</u>odel <u>D</u>eveloper of dynamic systems
  - Army Research Development and Engineering Command
  - User directed
  - Abstract base class hierarchy ♣

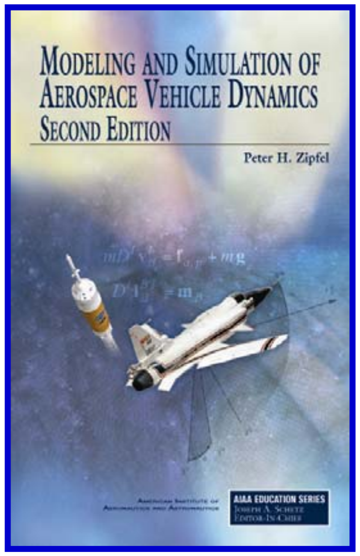
Comparison

	CADAC++	ENGAGE++	JSBSim	CMD
Purpose	Many aerospace environments, batch	3 DoF air-to-air engagements, batch GUI	Aircraft 6 DoF simulator, batch & realtime	Differential equation solver, batch & realtime
Class Structure	Abstract base class hierarchy	Partial hierarchical structure	Partial hierarchical structure	Abstract base class hierarchy
Components	Class functions	Derived classes	Derived classes	Derived classes
Interface	Protected data array, Combus	Parameter lists and public members	Get and set methods for private/protected data	Constructors and get/set methods
States	In-line integration	Central integration of public variables	In-line integration	Central integration in kernel
Events	By input file, frame implementation	GUI directed, in-line implementation	In-line 'switch' functions	User supplied stage vector with criteria
main()	execute.cpp	Not found	JSBSim.cpp	User supplied
Input	Input data and initialization by ASCII file	GUI	Set and get methods from input files	Constructors
Data	Data decks, input file	GUI	XML script files	Data files passing values to constructors
Output	Console, plot files	Launch envelopes, trajectories	Console, plot files	User supplied

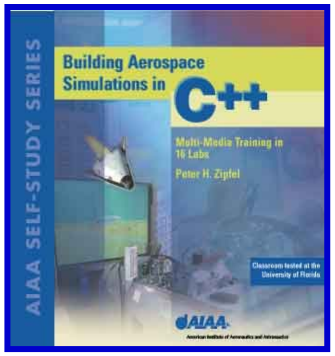
Summary

- Object oriented programming (OOP) is well suited to build netcentric aerospace simulations
- CADAC++ uses hierarchical structure with abstract base class
- Aerospace vehicles are modeled as derived classes
- Component modules are class functions
- Vehicle objects of CADAC simulation
  - 5 DoF model of turbojet driven bank-to-turn vehicle
  - Ground target, possibly moving
  - Satellite in circular or elliptical orbit
- UAV waypoint guidance
- Targeting by UAV seeker or satellite
- Availability: Open source ♣

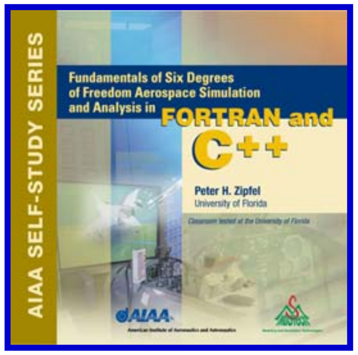
Author’s Resources from AIAA



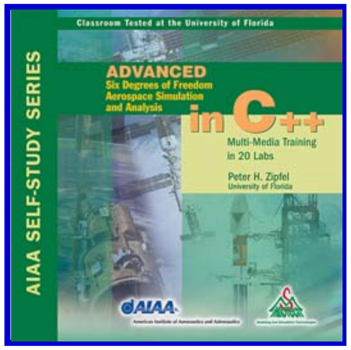
Modeling and Simulation of Aerospace Vehicle Dynamics, 2nd Edition, 2007



Building Aerospace Vehicle Simulations in C++, 2003



Fundamentals of Six DoF Aerospace Simulation and Analysis in FORTRAN and C++, 2004



Advanced Six DoF Aerospace Simulation and Analysis in C++, 2005

Textbook for graduate course in flight mechanics and M&S

Cruise missile source code, low fidelity

Missile and aircraft source code, high fidelity

Hypersonic vehicle source code, high fidelity